void *memset(void *s, int ch, size_t n);
函数解释：将s中当前位置后面的n个字节 （typedef unsigned int size_t ）用 ch 替换并返回 s 。

problem：当%c格式的时候，会读取任何字符，包括换行和空格。
当其他格式的时候版(不包括正则表达式)， 如果空格或者换行出现在前面，会被读取并抛弃

solve：scanf()函数在接收字符串时，遇到空格就会停止接收。可以使用gets()函数代替,然而系统提示gets不安全。

```
    gets(buf);
    ^
/tmp/ccXNRLdr.o: In function `main':
simple_shell.c:(.text+0x52): warning: the `gets' function is dangerous and shoul
d not be used.
```

可以在scanf()中使用正则
scanf("%[^\n]",buf);

ps：这里主要介绍一个参数，%[ ]，这个参数的意义是读入一个字符集合。[ ]是个集合的标志,因此%[ ]特指读入此集合所限定的那些字符，比如%[A-Z]是输入大写字母，一旦遇到不在此集合的字符便停止。如果集合的第一个字符是“^”，这说明读取不在“^“后面集合的字符，即遇到”^“后面集合的字符便停止。此时读入的字符串是可以含有空格的。（\n 表示换行符）

在进程的创建上Unix采用了一个独特的方法，它将进程创建与加载一个新进程映象分离。这样的好处是有更多的余地对两种操作进行管理。

当我们创建了一个进程之后，通常将子进程替换成新的进程映象，这可以用exec系列的函数来进行。当然，exec系列的函数也可以将当前进程替换掉。

例如：在shell命令行执行ps命令，实际上是shell进程调用fork复制一个新的子进程，在利用exec系统调用将新产生的子进程完全替换成ps进程。

exec系列函数（execl、execlp、execle、execv、execvp）
包含头文件<unistd.h>
功能：
　　用exec函数可以把当前进程替换为一个新进程，且新进程与原进程有相同的PID。exec名下是由多个关联函数组成的一个完整系列，
头文件<unistd.h>
extern char **environ;
原型：
```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);、
```

·这些函数如果调用成功则加载新的程序从启动代码开始执行，不再返回。
·如果调用出错则返回-1
·所以exec函数只有出错的返回值而没有成功的返回值。

execvp有两个参数：要运行的程序名和那个程序的命令行参数。当程序运行时命令行参数以argv[]传给程序。最后一个参数必须为NULL(即参数以0结束)，eg：

```
#include<stdio.h>
main(){
    char *arglist[3];
    arglist[0]="ls";
    arglist[1]="-l";
    arglist[2]=0;   //参数字符串必须以0结束
    printf("********About to execute ls -l\n");
    execvp("ls",arglist);
    printf("********ls is done.bye\n");
}
```

选择execvp的原因：

首先说参数传递。之所以选择execvp这个函数是因为该函数可以传递变长参数，也就是说，虽然参数列表度中只是两个指针变量，但由于第二个指针变量是变长指针数组，第二个参数传入参数个数实际是该数组的长度，可以通过控制指针数组的长度来控制传入的参数数量。

替换地址空间，实则将原进程的代码段，数据段进行替换，并未创建新的进程出来。

- 带 p 的exec函数：execlp,execvp，表示第一个参数path不用输入完整路径，只有给出命令名即可，它会在环境变量PATH当中查找命令
- 看看execvp()函数的API, 里面讲得是这样的, 它的度第一个参问数代表它要执行文件的位置, 第二个参数是命令(即命令+参数，如{ “ls”, "-l" }) execvp()搜索的PATH环境变量中答指定的目录专中的ls命令的位置，而传递参数属的ls命令在argv中

isspace() 返回值为非零表示c是空白符，返回值为零表示c不是空白符。
包含头文件<ctype.h>

status是否遇到新参数地标志

```
perror(): void perror(const char *s);

The  perror() function produces a message on standard error describing the last error
encountered during a call to a system or library function.
```

```
#include<sys/types.h>
#include<sys/wait.h>
pid_t wait(int* status);
```

返回值：
成功返回被等待进程pid，失败返回-1。
参数：
输出型参数，获取子子进程退出状态，不关心心则可以设置成为NULL

僵尸进程是当子进程比父进程先结束，而父进程又没有回收子进程，释放子进程占用的资源，此时子进程将成为一个僵尸进程。

一个进程在调用exit命令结束自己的生命的时候，其实它并没有真正的被销毁， 而是留下一个称为僵尸进程（Zombie）的数据结构（系统调用exit，它的作用是 使进程退出，但也仅仅限于将一个正常的进程变成一个僵尸进程，并不能将其完全销毁）

```
wait(NULL);等待子进程结束，回收子进程
```

函数名： dup2

功能： 复制文件描述符

用法： int dup2(int oldfd,int newfd);

“输出重定向”的功能可以用 dup2(fd ,1) 。或者是dup(fd, STDOUT_FILENO)

```
32
33  #define STDOUT_FILENO 1
34
```

- dup2 与 dup 区别是 dup2 可以用参数 newfd 指定新文件描述符的数值。

我们知道，每一个进程都有3个标准的输入输出文件描述符

描述符编号	简介	作用
0	标准输入	通用于获取输入的文件描述符
1	标准输出	通用输出普通信息的文件描述符
2	标准错误	通用输出错误信息的文件描述符

http://blog.chinaunix.net/uid-31433594-id-5761435.html

problem为啥执行完close(fd)后还能接收终端输出?

```
                }
                argv[i] = NULL;

                // create or open file
                int fd =open(argv[i+1], O_RDWR|O_CREAT|O_TRUNC, 0664);
                printf("open file: %s\n", argv[i+1]);
                if(fd == -1){
                        perror("open");
                        exit( 1);
                }

               // redirct
                printf("receive data1\n");
                dup2(fd, 1);
                printf("receive data2\n");
                close(fd);
                printf("receive data3\n");
            }
        }

        execvp(buf, argv);
        perror("fork");
        exit(1);
}
else{
        printf("father\n");
```

同时存在管道和重定向时，优先级问题?

将可执行文件添加到系统环境

```
lyq@lyq-virtual-machine:~/Documents/lab3$ export PATH=/usr/bin:/bin:/home/lyq/Do
cuments/lab3:$PATH
lyq@lyq-virtual-machine:~/Documents/lab3$ cmd1
excute cmd1
lyq@lyq-virtual-machine:~/Documents/lab3$ ./test
father
child
excute cmd1
lyq@lyq-virtual-machine:~/Documents/lab3$ ls
1.txt  cmd1.c  cmd2.c  cmd3.c  simple_shell.c  test.c
cmd1   cmd2    cmd3    shell   test
lyq@lyq-virtual-machine:~/Documents/lab3$ ./shell
shell# cmd1
buf:cmd1
excute cmd1
shell# cmd3 aa aa
buf:cmd3
argc: 3
result: aaaa
```

```
lyq@lyq-virtual-machine:~/Documents/lab3$ export PATH=/home/lyq/Documents/lab3:/usr/bin:/bin:$PATH
lyq@lyq-virtual-machine:~/Documents/lab3$ ls
1.txt  cmd1  cmd1.c  shell  simple_shell.c  test  test.c
lyq@lyq-virtual-machine:~/Documents/lab3$ cmd1
excute cmd1
```

管道

Shell中通过`fork`+`exec`创建子进程来执行命令。如果是含管道的Shell命令，则管道前后的命令分别由不同的进程执行，然后**通过管道把两个进程的标准输入输出连接起来**，就实现了管道。

在Shell中要实现管道这样的效果，有4个步骤：

1. 创建pipe
2. fork两个子进程执行管道前后的命令
3. 把第一个子进程(前者)的标准输出重定向到管道数据入口
4. 把第二个子进程(后者)的标准输入重定向到管道数据出口